Efficient Summary Reuse for Software Regression Verification

Fei He, Qianshan Yu, and Liming Cai

Abstract—Software systems evolve throughout their life cycles. Many revisions are produced over time. Verifying each revision of the software is impractical. Regression verification suggests reusing intermediate results from the previous verification runs. This paper studies regression verification via summary reuse. Not only procedure summaries, but also loop summaries are proposed to be reused. This paper proposes a fully automatic regression verification technique in the context of CEGAR. A lazy counterexample analysis technique is developed to improve the efficiency of summary reuse. We performed extensive experiments on two large sets of industrial programs (3,675 revisions of 488 Linux kernel device drivers). Results show that our summary reuse technique saves 84% to 93% analysis time of the regression verification.

Index Terms—Regression Verification, Program Verification, Abstraction Refinement, Summary Reuse.

1 Introduction

Along with the widespread use of software in our daily life, there is a growing concern for software reliability. At the same time, market pressure demands quick product introductions. The software companies are required to introduce new features to their software products in shorter release cycles. Since errors may be introduced with new features, the new products must be reverified to ensure their correctness.

Software verification [1] has made great success in recent years. However, it is still very time-consuming. Verifying every revision of the software is impractical. Inspired by the success of regression testing [2], [3], researchers in formal verification community proposed the technique of regression verification [4], [5], [6], [7], [8]. Taking into consideration that many intermediate results are produced during the verification, and the computation of these results is costly, regression verification aims to make use of these intermediate results in the verification of new program revisions.

Different intermediate results have been proposed for reuse, including abstract precisions, state-space graphs, constraint solver solutions, and interpolation-based procedure summaries. Beyer et al. [7] proposed to record the final abstract precision in the previous verification run, and reuse it in the current verification. Henzinger et al. [9] proposed to reuse the state-space graph for incremental checking of temporal safety properties. Visser et al. [10] noticed the important role of constraint solving in software verification, and proposed to reuse the constraints solving results.

Procedure summaries, representing input/output behaviors of procedures, have been proposed in [11] to be reused in incremental upgrade checking. Note that procedure summaries are reasonably small to store, technically easy to process, and do not require much extra computation effort to be reused. Therefore, reusing procedure summaries is a good choice for regression verification.

Inspired by [11], this paper studies the summary-based regression verification for predicate analysis. In [11], the procedure summaries are mainly constructed by interpolations. In this paper, we consider the summaries constructed using abstract states of predicate analysis. Note that these abstract states are by-products of program analysis [12], [13], [14]. Thus, it does not require additional computational effort to generate these summaries. Moreover, different from existing techniques, our approach considers the reuse of not only procedure summaries, but also loop summaries. We build a unified framework for reusing both of them.

Moreover, we consider regression verification in the context of counter-example guided abstraction refinement (CEGAR) [15]. Summary reuse techniques need to be adapted to the CEGAR framework (see Section 5). A lazy counterexample analysis technique is further proposed to address the effectiveness issue of summary reuse (see Section 5.3). Considering that CEGAR is a widely-adopted technique in software verification [16], [17], [18], [19], [20], our approach can be applied to most state-of-the-art software verifiers. To the best of our knowledge, our approach represents a novel attempt to the regression verification with CEGAR.

We implemented our approach on top of CPAchecker [17]. We have performed extensive experiments on two large sets of industrial programs. The first set of programs contains 1,119 real-world program revisions of 62 Linux device drivers, and the second contains 2,556 artificial program revisions (by mutation) of 426 Linux device drives. In total, there are 6,749 verification tasks, among which 6,064 are regression verification tasks. Experimental results show a very promising performance of our approach. With the set of real-world programs, in comparison to the standalone verification without reuse, our approach solves 216 more regression verification tasks and saves 93.1% of analysis time. With the set of artificial programs, our approach solves 10 more regression verification tasks and saves 84.2% of analysis time.

• *Fei He, Qianshan Yu and Liming Cai are with the School of Software, Tsinghua University, Beijing 100084, China.*
E-mail: hefei@tsinghua.edu.cn, yuqianshan@foxmail.com

Manuscript received XXXX, 2019; revised XXXX, XXXX.

The main technical contributions of this paper are summarized as follows:

- We propose a unified framework for reusing both procedure summaries and loop summaries.
- We propose a fully automatic regression verification technique in the context of counterexample-guided abstraction refinement. A novel lazy counterexample analysis technique is developed to improve the efficiency of summary reuse.
- We implement our approach in the software verification tool CPAchecker . Experimental results show the promising performance of our approach.

The remainder of this paper is organized as follows. Section 2 introduces the necessary backgrounds. Section 3 motivates our approach using a simple example. Section 4 reviews the CEGAR-based program verification and the definition of program summaries. Section 5 presents our CEGAR-based regression verification framework. Section 6 reports evaluation results on our approach. Section 7 discusses related work and Section 8 concludes this paper.

2 Backgrounds

2.1 Abstraction and Refinement

Abstraction plays a central role in software verification. Abstraction omits details of the system behaviors, resulting in a simpler model. We call the model before and after abstraction the *concrete* and the *abstract* model, respectively. An abstraction is *conservative* [21] iff it does not omit any behavior of the concrete model. Conservative abstraction guarantees that the properties (more precisely, the $ACTL^*$ properties [21]) established on the abstract system also hold on the concrete system. The reverse, however, is not guaranteed: if the abstract model falsifies the property, the concrete model does not necessarily falsify this property.

The *abstract precision* [7] (for short, *precision*) defines the level of abstraction of an abstract model. The precision must be at a proper level. A too-coarse precision may fail to verify the property; a too-fine precision, however, may lead to state space explosion. Finding a proper precision appears to require ingenuity.

Counterexample-guided abstraction refinement [15] provides a framework for automatically finding proper precisions. Starting from an initial abstract precision, it iteratively checks if the corresponding abstract model satisfies the desired property. If the property is satisfied, it must also hold on the concrete model, the algorithm terminates and reports "correct". Otherwise, the checker returns a path on the abstract model that falsifies the desired property. The algorithm then checks if the returned path is valid on the concrete model or not. If it is, the algorithm finds a real bug, it thus terminates and reports "incorrect". Otherwise, the precision is too coarse, and needs to be refined with the counterexample. Then the above process repeats, until either "correct" or "incorrect" is reported.

The abstract precision does not necessarily keep the same throughout the program [22]. To simplify the discussion, we assume in this paper that the abstract precisions are defined at the level of procedures, i.e., each procedure is associated with a unique abstract precision.

2.2 Software Verification

Model checking and program analysis are two major approaches for software verification. Comparing these two techniques, model checking is more precise with fewer false positives produced, while program analysis is comparatively more efficient and can be applied to more programs. An increasing tendency to software verification is to integrate these two techniques together [23], to get a good balance between accuracy and efficiency.

Predicate abstraction [22], [24] is a widely-adapted abstraction technique [1], [19] for software verification. It creates an abstract model with respect to *a set of predicates* defined on the program variables. This predicate set defines an abstract precision for predicate abstraction. The state space of the abstract model is only related to the number of predicates in the abstract precision. Finding proper predicates is the key problem for predicate abstraction. One popular technique is based on interpolation computation on the counterexamples [25], [26].

Interprocedural analysis deals with programs with multiple procedures. One simple way of interprocedural analysis is to inline a copy of the callee procedure at each of its call sites. The inlining technique is, however, very expensive and may lead to context explosion for recursive procedures. Another interprocedural analysis technique is to use summaries [12], [13]. A procedure summary (or shortly, a summary) describes the input/output behaviors of a procedure. This technique plugs summaries at each call site of the procedure. Re-analysis of the procedure body at each of its call sites can be avoided using this technique, the efficiency is therefore improved.

There are at least two kinds of procedure summaries in literature: the state-based summaries [12], [13], where each summary is a pair of input and output states of the procedure; and the interpolation-based summaries [11], where the summary is an overapproximation of the procedure's behaviors.

In this paper, we assume a deterministic, single-threaded program and a safety property. To specify the property, a special `error` location is introduced in the program. We say the program is *correct* if and only if the `error` location is not reachable.

3 A Motivating Example

Figure 1 shows a simple program that consists of two procedures: `main` and `inc`. A `while` loop is implemented in the `main` procedure, and in the loop body the `inc` procedure is invoked. The `inc` procedure takes two input parameters: a and $sign$, and outputs either $a + 1$ (if $sign! = 0$), or $a - 1$ (if $sign = 0$). We want to verify that the `error` location (at line 6) is not reachable in any execution of this program.

Consider an invocation to the `inc` procedure (at line 3) with parameters $a = 0$ and $sign = 1$, the returned value is $rv = 1$. The pair of this entry state (i.e. $a = 0 \wedge sign = 1$) and the exit state (i.e. $rv = 1$) summarizes this execution of the `inc` procedure. Later, when the `inc` procedure is invoked again, if its entry state is again $a = 0 \wedge sign = 1$, then without entering the `inc` procedure, we can immediately determine its exit state as $rv = 1$.

```
  main()
  {
     int i = 0, x = 0;
1:   while(i < 10) {
2:     if(x <= 5)
3:        x = inc(x, 1);
4:     else
          x = inc(x, 0);
5:     i = inc(i, 1);
     }
6:   if(x < 5) goto error;
  }

  inc(int a, int sign)
  {
7:   if (sign) return a + 1;
8:   else return a - 1;
  }
```

Fig. 1. An example program

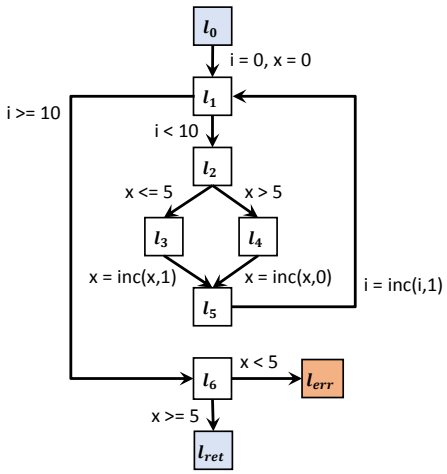


Fig. 2. CFA of the `main` procedure in Fig. 1

The execution of a loop can also be summarized by a pair of an entry state and an exit state. Consider the `while` loop in the `main` procedure, its entry state (i.e. the state exactly before the program enters the loop at line 1) is $i = 0 \wedge x = 0$, and its exit state (i.e., the state when the program exits the loop at line 6) is $x = 6 \wedge i = 10$. Similar to the procedure summary, the pair of these two states also summarizes an execution of this loop, and is called a *loop summary*.

Assume that the original program evolves to a new revision. Apparently, this new revision needs also to be checked to guarantee its correctness. Assume that in the new revision, the `inc` procedure does not change, then the summaries of this procedure, which were generated in the previous round of verification, can be reused in the new round of verification. Similarly, if the `while` loop does not change in the new revision, the previous-generated summaries for this loop can also be reused. *How to efficiently reuse the previously-generated summaries in regression verification* is the main research problem we want to solve in this paper.

Moreover, in the above discussions, the program is analyzed by tracking its concrete states. The concrete state space of a program is, however, considerably huge and often infinite. A practical verification technique (including the regression verification) needs to be performed on the abstract state space. *How to efficiently combine regression verification and abstraction techniques, especially the counterexample-guided abstraction refinement*, is another research target of this paper.

4 CEGAR-BASED VERIFICATION

In this section, we first review the CEGAR-based program verification, upon which our regression verification scheme is based. We then propose a unified definition for procedure summaries and loop summaries.

4.1 Preliminaries

We begin by introducing the necessary preliminaries for program verification.

Control-Flow Automata (CFA) [22], [23] were adopted in many software verification techniques (for example, BLAST and CPAchecker) for representing programs. Given a program P, let $\mathcal{L}$ be the set of program locations and St be the set of statements of P, respectively. The CFA of P is a pair $(\mathcal{L}, G)$, where $\mathcal{L}$ is the set of program locations, and $G \subseteq \mathcal{L} \times St \times \mathcal{L}$ is the set of control flow edges. The CFA is different from the control flow graph with program statements labeling the edges rather than the vertices. For example, CFA of the `main` procedure in Fig. 1 is shown in Fig. 2, where l_{err} represents the error location, and l_0, l_{ret} represent the entry and exit locations of the `main` procedure, respectively.

A *state* of a program is a configuration of the program location and the set of facts that we know about the program at that location. Formally, a *concrete state* of the program P is a pair (l, u), where $l \in \mathcal{L}$ is a program location and u is a full assignment to all variables of P. The assignment u is also called the *concrete data state* of P. Let λ be a set of predicates, representing the current abstract precision. An *abstract state* is a pair (l, s), where l is a program location, and s is a valuation to all predicates in λ. The valuation s is also called the *abstract data state* of P. In the remainder of the paper, we denote P^λ the abstract model of P with respect to λ.

Consider the CFA of the `main` procedure in Fig. 2, with the abstract precision $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$. An abstract data state is a valuation to the three predicates in λ_{main}. During the procedure of the analysis, the value of a predicate may be $true$ (abbreviated by 0), $false$ (abbreviated by 1) or non-deterministic (abbreviated by $*$). We use a vector to denote an abstract data state. For example, the abstract data state at l_0 is $[*, *, *]$ (for short, written $***$), indicating that all predicates' values are non-deterministic at this location. And when the program transits from l_0 to l_1, the abstract data state at l_1 is 111, since executing the statements `i=0` and `x=0` can make the three predicates all $true$.

A *path* π of the program is an alternating sequence of states and program statements, i.e.,

$$\pi = (l_0, s_0) \xrightarrow{st_0} (l_1, s_1) \xrightarrow{st_1} \ldots \xrightarrow{st_{n-1}} (l_n, s_n).$$

A path π is a *concrete path* of P (or an *abstract path* of P^λ) iff all states on π are concrete states of P (or abstract states of P^λ). A path π is called a *CFA path* if l_0 is the entry location of the program, and for each i with $0 \leq i < n$

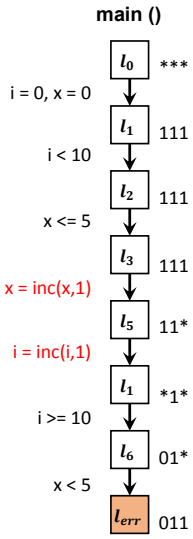


Fig. 3. A counterexample of the program in Fig. 1 with $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$

there exists a CFA edge $g = (l_i, st_i, l_{i+1})$. In other words, a CFA path represents a syntactical walk through the CFA. A *counterexample* of P (or P^λ) is a CFA path of P (or P^λ) that ends at the `error` location.

Consider the CFA in Fig. 2 with the abstract precision $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$. A possible counterexample is shown in Fig. 3, where the abstract data states are labelled beside the corresponding program locations.

We use the *strongest post-condition operator* SP to define the semantics of a CFA path. For a formula φ and a statement st, $SP_{st}(\varphi)$ represents the set of data states that are reachable from any of the states that satisfy φ after the execution of st. Let $st_0, st_1, \ldots, st_{n-1}$ be the sequence of program statements passed by the CFA path π. The *semantics* of π is the successive application of the SP operator to each statement of π, i.e., $SP_\pi(\varphi) = SP_{st_{n-1}}(\ldots(SP_{st_0}(\varphi)\ldots)$.

Definition 1. A CFA path π starting from the abstract state (l, s) is *feasible* iff $SP_\pi(s)$ is satisfiable.

Note that a feasible path is always a CFA path. Let (l_0, s_0) be the initial state of P^λ. An abstract state (l, s) of P^λ is *reachable* iff there exists a feasible path π of P^λ that ends at the location l such that $s \models SP_\pi(s_0)$.

Definition 2. The abstract model P^λ is *correct* iff the `error` location is not reachable in P^λ.

4.2 CEGAR

We next describe the scheme of the standalone program verification (i.e., without reuse) via predicate abstraction and CEGAR [22], [23].

Let λ be an abstract precision, and P^λ be the abstract model with respect to λ. Since the predicate abstraction is conservative [24], to verify the program P, it is sufficient to find a proper abstract precision λ such that P^λ is correct. This can be achieved by the scheme of the counterexample-guided abstraction refinement (Fig. 4). Initially, the abstract

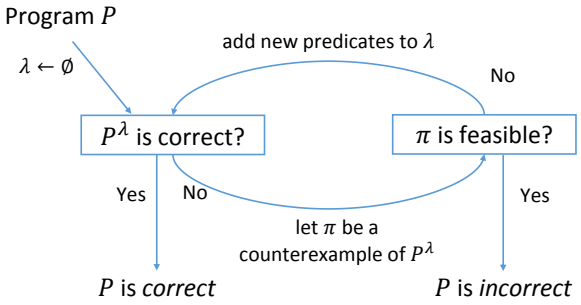


Fig. 4. CEGAR-based verification

precision is set to empty. The abstract precision is then iteratively refined by adding new predicates, until the program is verified.

Each iteration consists of two phases: a model validation phase and a counterexample analysis phase. During the model validation phase, we check if the abstract model P^λ is correct or not. If P^λ is correct, we immediately conclude that P is also correct. Otherwise, we get a counterexample π that is a CFA path of P^λ ending at the `error` location.

During the counterexample analysis phase, the counterexample π is semantically analyzed to determine whether it is feasible or not. If it is feasible, we find a real execution of the program P that reaches `error`, and we thus conclude that P is incorrect. Otherwise, π is a spurious counterexample, and the proof of its infeasibility can be used to refine the abstract precision [22]. The refinement is performed by adding new predicates in the abstract precision, such to eliminate the spurious counterexample from the refined model. After the refinement, the next iteration continues.

4.3 Summaries

We now introduce a unified definition for the procedure and loop summaries.

Let ϱ be a program fragment (either a procedure or a loop). An *entry state* of ϱ is a state at the entry location of ϱ, and an *exit state* of ϱ is a state at its exit location. A pair of an entry state and an exit state summarizes the input/output behavior of one execution of ϱ [12], [13].

Definition 3. A *summary* of a program fragment ϱ is a triple $\langle \lambda, \phi_{in}, \phi_{out} \rangle$, where λ is an abstract precision, ϕ_{in} and ϕ_{out} are Boolean combinations of predicates in λ, representing an entry state and a set of exit states of ϱ, respectively.

A summary states that if the entry state of ϱ satisfies ϕ_{in}, its exit state must satisfy ϕ_{out}. This definition is particularly suitable for predicate analysis. Let λ be the current abstract precision, and P^λ be the abstract model of P with respect to λ. The model validation is essentially to traverse the state space of P^λ [17] to find that if the `error` location is reachable or not. Let (l, s) be the current abstract state, and ϱ be the program fragment to be executed, the model validation algorithm needs to traverse all possible paths of ϱ (under the abstract precision λ) to compute the set of exit states. Let ϕ be the formula representing the entry state (l, s), and ϕ' be the formula representing the set of exit states, the triple $\langle \lambda, \phi, \phi' \rangle$ is a summary of ϱ.

With the above definition, a summary corresponds to a subset of paths in ϱ. The main advantage of using the state-based summaries is the efficiency. Consider a state-based summary $\langle \lambda, \phi_{in}, \phi_{out} \rangle$, the abstract precision λ is determined at each iteration of CEGAR; the entry state ϕ_{in} and the set ϕ_{out} of exit states are computed in the model validation process. In conclusion, all ingredients of this summary are by-products of CEGAR. There needs no additional computation to generate the state-based summaries.

Note that an abstract precision must be specified in the summary, since the entry state and the exit state must both be defined over the predicates in the abstract precision. Recall that we assume a unique abstract precision throughout a procedure or a loop. Thus only one abstract precision needs to be specified here. Otherwise, if the abstract precision differs in different points of ϱ (for example, as in the lazy abstraction [22]), we need to specify an abstract precision for ϕ_{in} and ϕ_{out}, respectively.

All the generated summaries are maintained in a *summary cache* Ξ. During the program analysis, whenever a program fragment is encountered, the verifier seeks in Ξ for an applicable summary. Let λ_c be the current abstract precision, and s_c be the current abstract data state. A summary $\langle \lambda, \phi_{in}, \phi_{out} \rangle$ of ϱ is called *applicable* iff $\lambda_c \subseteq \lambda$ and $s_c \Rightarrow \phi_{in}$. If any applicable summary exists, the verifier directly uses ϕ_{out} of this summary as the exit state of ϱ. Otherwise, the verifier needs to conduct a heavy fix-point computation [27] on the fragment ϱ to compute its exit state.

5 CEGAR-based Regression Verification

Summaries convey important information about the verification. In this section, we propose some efficient summary reuse techniques for regression verification.

5.1 Overview

An overview of our CEGAR-based regression verification is shown in Fig. 5. Besides the program P, a set Ξ' of the previously-generated summaries is also provided for the regression verification. Note that these summaries are produced by the previous revisions, and may not be applicable to the current revision. Similar to [11], we propose a *summary selection* step to guarantee the safe reuse of summaries (see Section 5.2).

As in a standalone verification, each iteration of CEGAR for a regression verification also consists of two phases: a model validation phase and a counterexample analysis phase. The former phase is exactly the same as in the standalone verification. The *counterexample analysis*, however, requires more careful handling, which will be discussed in Section 5.3.

Note that our summary reuse does not depend on the verification result. A summary here represents an execution of the corresponding program fragment. No matter whether the verification result is "correct" or "incorrect", as long as the fragment does not change semantically, the summary can be reused. This is very different from the interpolation-based summaries [11], where the summaries are related to the property to be verified, and can only be reused when the verification result is "correct".

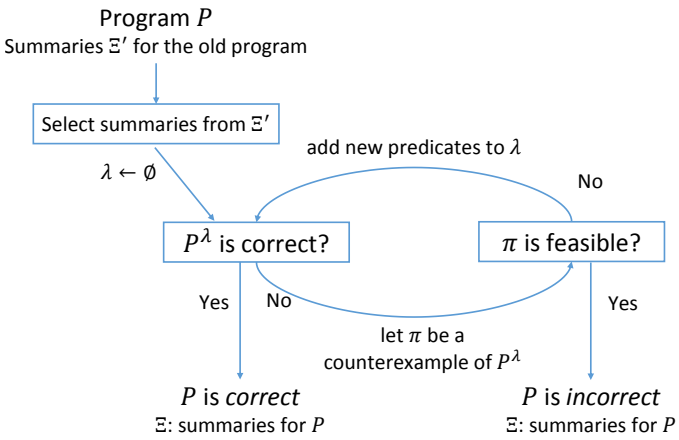


Fig. 5. CEGAR-based regression verification

forall $\varrho \in P$ **do**
  Let ϱ' be its previous version in P';
  if $\varrho \equiv \varrho'$ **then**
    Select summaries of ϱ';
  end
end
Use selected summaries to initialize Ξ ;

Algorithm 1: Summary selection

5.2 Summary Selection

Let P' be the old revision of P. For each fragment ϱ (either a procedure or a loop) of P, let ϱ' be its previous version in P'. If ϱ' does not exist in P', i.e., ϱ is a newly added fragment in P, we simply let $\varrho' = NULL$.

In the summary selection (Alg. 1), we check for each ϱ of P if ϱ is semantically equivalent to ϱ' or not. If it is, the summaries of ϱ' are selected. Otherwise, these summaries are abandoned. These selected summaries are then reused in the regression verification to initialize the summary cache Ξ.

Note that the semantic equivalence checking $\varrho \equiv \varrho'$ is very expensive in computation, we thus choose to perform syntactic checking instead. The syntactic checking is less precise, i.e., may miss some semantically equivalent fragments, but preserves the soundness, i.e., the fragments that pass the syntactic checking must be semantically equivalent.

5.2.1 Syntactic Checking

In the following, we first discuss two notions, i.e. syntactically unchanged fragments and semantically equivalent fragments, then, we propose our syntactic checking technique.

Definition 4. Let ϱ be a program fragment in P and ϱ' be its previous version in P', we say ϱ is *syntactically unchanged* if all statements of ϱ and ϱ' are same, and *syntactically changed*, otherwise.

In the following, we consider only syntactically unchanged fragments for possible summary reuse. Note that a syntactically changed procedure may still be semantically equivalent to its previous version. We ignore this case since the semantic equivalence checking is too expensive.

On the other hand, a syntactically unchanged fragment is not necessarily to be semantically equivalent. Comparing only statements in a fragment is not enough for checking the semantics of this fragment. For example, the `call` statement in a fragment may lead the execution to a statement outside the fragment (in the called procedure). If the called procedure changes, even all statements in the fragment under consideration remain the same, its semantics has changed.

Consider the program in Fig. 1, we assume that in the new program revision the `main` procedure remains the same while the `inc` procedure changes, then the semantics of the `main` procedure is regarded as changed.

To summarize, the syntactic changes of a fragment may lead the semantics of another fragment to be changed. We thus have the following definition.

Definition 5. Let ϱ_1, ϱ_2 be two program fragments of P, we say ϱ_1 *impacts* ϱ_2, written $\varrho_1 \prec \varrho_2$, if either of the following statements is satisfied:

- ϱ_1 is a procedure and is *called* in ϱ_2, or
- ϱ_1 is a loop and is *nested* in ϱ_2.

The above *impact* relation extends the call relation over procedures by taking loops and their nesting structures into consideration.

The impact relation is *transitive*, i.e., if $\varrho_1 \prec \varrho_2$ and $\varrho_2 \prec \varrho_3$, then $\varrho_1 \prec \varrho_3$. Let $\prec^*$ be the *transitive closure* of $\prec$. If $\varrho_1 \prec^* \varrho_2$, i.e., there exist $\varrho_i, .., \varrho_{i+k}$ such that $\varrho_1 \prec \varrho_i \prec \ldots \prec \varrho_{i+k} \prec \varrho_2$, we call ϱ_2 is *reachable* from ϱ_1. Moreover, we define the set of *forward reachable* fragments from ϱ_1 as $FReach(\varrho_1) = \{\varrho_2 \mid \varrho_1 \prec^* \varrho_2\}$. Apparently, $FReach(\varrho_1)$ is the *maximal* set of fragments that ϱ_1 can impact. We also define the set of *backward reachable* fragments from ϱ_1 as $BReach(\varrho_1) = \{\varrho_2 \mid \varrho_2 \prec^* \varrho_1\}$, which is the *maximal* set of fragments that have impact on ϱ_1.

Consider the three fragments in the program in Fig. 1: the `main` procedure (denoted as ϱ_{main}), the `inc` procedure (denoted as ϱ_{inc}) and the loop in the `main` procedure (denoted as ϱ_{loop}). They satisfy: $\varrho_{inc} \prec \varrho_{loop}$, $\varrho_{inc} \prec \varrho_{main}$ and $\varrho_{loop} \prec \varrho_{main}$. Thus we have $FReach(\varrho_{inc}) = \{\varrho_{loop}, \varrho_{main}\}$, $BReach(\varrho_{inc}) = \emptyset$. In other words, any change in ϱ_{inc} can impact the semantics of ϱ_{loop} and ϱ_{main}, and no other fragment can impact the semantics of ϱ_{inc}.

The concepts of forward/backward reachable sets can be lifted to a fragment set. Let τ be a set of fragments, $FReach(\tau) = \bigcup_{\varrho \in \tau} FReach(\varrho)$, and $BReach(\tau) = \bigcup_{\varrho \in \tau} BReach(\varrho)$.

Definition 6. Let ϱ be a program fragment in P and ϱ' its previous version in P', we say ϱ is *globally syntactically unchanged* if

1) ϱ is syntactically unchanged, and
2) all fragments in $BReach(\varrho)$ are syntactically unchanged.

Lemma 1. A globally syntactically unchanged fragment is semantically equivalent to its previous version.

Proof: This is a direct conclusion by the definition of globally syntactically unchanged fragment. □

To find the globally unchanged fragments of P, we syntactically compare each fragment of P to its previous version. According to the comparing results, fragments in P are divided into two parts: the syntactically unchanged set τ_1 and the syntactically changed set τ_2. Then we have the following lemma.

Lemma 2. Let τ_1 and τ_2 be the set of syntactically unchanged and syntactically changed fragments of P, respectively. Fragments in $\tau_1 \setminus FReach(\tau_2)$ are all globally syntactically unchanged.

Proof: Assume that the lemma does not hold, i.e., there is a fragment $\varrho_1 \in \tau_1 \setminus FReach(\tau_2)$ that is not globally syntactically unchanged. By $\varrho_1 \in \tau_1$ and Definition 6, there must be a fragment $\varrho_2 \in BReach(\varrho_1)$ such that ϱ_2 is syntactically changed. By $\varrho_2 \in BReach(\varrho_1)$, we have $\varrho_1 \in FReach(\varrho_2)$. By ϱ_2 being syntactically changed, we have $\varrho_2 \in \tau_2$, and thus $\varrho_1 \in FReach(\tau_2)$. This is contradicted with the assumption. Thus the assumption does not hold, and the lemma holds. □

Let $\tau^* = \tau_1 \setminus FReach(\tau_2)$. The semantic equivalence checking $\varrho \equiv \varrho'$ (on Row 4 of Alg. 1) is implemented as checking whether $\varrho \in \tau^*$. If $\varrho \in \tau^*$, the summaries of ϱ are selected, and otherwise they are abandoned. Note that the computations of τ_1, τ_2 and τ^* involve only syntactic checking of P and P'. According to Lemma 1 and Lemma 2, all selected summaries are semantically equivalent to its previous version, and thus they can be safely reused in the regression verification.

5.3 Counterexample Analysis

Given a counterexample returned by the model validation process, we need to check if this counterexample corresponds to a real bug or not. Summary reuse makes this process intricate.

Consider the counterexample in Fig. 3 that contains two procedure calls. During the program verification, these two procedure calls are replaced by two *abstract* summaries, which are defined over predicates and may introduce spurious behaviors over the program's concrete semantics. Therefore, to check the feasibility of this counterexample, the inner paths in the `inc` procedure that correspond to these two summaries must be restored.

Consider the procedure call `inc(x,1)` between $(l_3, 111)$ and $(l_5, 11*)$ for example. If the summary for this procedure call is generated in the current verification run, the inner path in `inc` that leads from $(l_3, 111)$ to $(l_5, 11*)$ is available [17]; otherwise, if the summary is inherited from the previous verification, there is no information for the inner path. Then, we have to rely on the heavy fix-point computation [27] to reproduce this path. In other words, with the counterexample checking, the saved analysis on the `inc` procedure is getting back. The benefits of summary reuse are thus significantly weakened.

5.3.1 Holes

Definition 7. A summary on a path is called a *hole* if it is inherited from the previous verification runs.

Let π be a counterexample path with holes. Replacing a hole with the corresponding inner path is called an *expansion*. For example, Fig. 6 shows the expanded version of the counterexample in Fig. 3. We call a path *holeless* if it contains no hole.

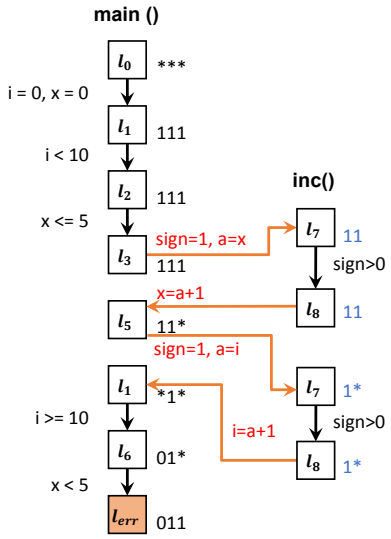


Fig. 6. An expanded version of the counterexample in Fig. 3, with $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$ and $\lambda_{inc} = \{sign > 0, a < 5\}$

Let H be the set of holes on a path π. The path π is split by these holes into $|H| + 1$ *path segments*. Each of these segments is a *holeless* path. The semantics of a path with holes is defined as the conjunction of the semantics of its holeless segments.

Theorem 1. Let π be a path with holes H, and Π the set of segments of π split by H,

1) if there exists any infeasible segment in Π, π is infeasible; and
2) if all segments in Π are feasible, π is, however, not necessarily feasible.

For the latter case, if all segments of π are feasible, we call the path π *separately feasible*. The above theorem states that the separate feasibility does not imply the feasibility of the whole path. This is obvious since the semantics of holes are not taken into consideration in the separate feasibility.

Consider the counterexample in Fig. 3, the holes at l_4 and l_5 split the path into three segments, i.e.

$$\sigma_1 : (l_0, ***) \rightarrow (l_1, 111) \rightarrow (l_2, 111) \rightarrow (l_3, 111),$$
$$\sigma_2 : (l_5, 11*),$$
$$\sigma_3 : (l_1, *1*) \rightarrow (l_6, 01*) \rightarrow (l_{err}, 011).$$

This counterexample is infeasible if any of σ_1, σ_2 and σ_3 is infeasible. Reversely, even σ_1, σ_2 and σ_3 are all feasible, the path is not necessarily feasible since the inner paths from l_3 to l_5 and from l_5 to l_1, that are replaced by summaries, may be infeasible.

5.3.2 Lazy Analysis Algorithm

A *brute-force* algorithm for counterexample analysis is to directly expand all holes of the counterexample, and then check its feasibility. This algorithm is correct but inefficient. Recall that to expand a hole, we need to perform a heavy analysis on the program fragment, which usually costs lots of computation. In the worst case, it needs to traverse all paths of the fragment to reproduce the path from a given input state to a given output state. We therefore propose a technique to avoid unnecessary hole expansions.

Input: A finite abstract path π
Output: The expanded path of π if it is infeasible; or "$unsafe$" if it corresponds to a real path.

while $\neg(isHoleless(\pi) \vee isInfeasible(\pi))$ **do**
  let h be a hole in π;
  $\pi \leftarrow expandHole(\pi, h)$;
end
if $isInfeasible(\pi)$ **then return** π ;
else return $incorrect$;

Algorithm 2: $lazyAnalysis(\pi)$

Our *lazy analysis* algorithm is shown in Alg. 2. The basic idea is to expand holes on demand, so as to avoid unnecessary hole expansions. The main body of the algorithm is a *while* loop. At the beginning of each iteration of the loop, the algorithm checks whether the current path is holeless ($isHoleless(\pi)$), and whether the current path is infeasible ($isInfeasible(\pi)$). If both checks return $false$, the loop continues by expanding one hole in π. Otherwise, the current path π must be either infeasible or holeless. For the former case, the algorithm returns π; and for the latter case, the algorithm reports "$incorrect$".

Comparing to the brute-force approach, our lazy algorithm needs more feasibility checking. However, it is still beneficial. Firstly, with the lazy approach, the computational efforts for unnecessary hole expansions (which in many cases are expensive) are saved. Secondly and more importantly, the returned path by the lazy approach is often much shorter than the fully expanded one. Note that the refinement is a heavy step in CEGAR [26]. With a shorter counterexample, the computation efforts for the refinement (for example, the interpolation-based refinement [26]) can often be significantly reduced.

The lazy analysis algorithm can be easily adapted to the existing CEGAR framework (for example, CPAchecker [17]) in the following way. When the verifier in the existing framework returns a counterexample, our algorithm is applied to check if this counterexample is spurious or not. In case of a spurious counterexample, our algorithm returns a (partially) expanded path and gives it to the existing refiner in the framework. The returned path by our algorithm may contain holes. Treating these holes as value assignments, these paths can be directly processed by most of the existing refinement techniques, for example, the interpolation-based refinement [26].

5.4 Precision Reuse

In the beginning of the regression verification (in Fig. 5), the abstract precision is set to be empty, which is in fact not necessary. Dirk Beyer et al. [7] showed that the abstract precision can also be reused in the regression verification. To take this idea, we simply use the final abstract precision λ' in the previous verification to initialize the current abstract precision λ, i.e., replace $\lambda \leftarrow \emptyset$ with $\lambda \leftarrow \lambda'$ in Fig. 5.

With precision reuse, the amount of summaries that need be recorded at the end of the verification run is also reduced. Assume that the abstract precision of the regression verification is initialized as λ'. Then during the regression verification, the abstract precision is iteratively refined by

CEGAR. In other words, summaries with a smaller abstract precision than λ' are useless in the regression verification. Thus, we need only to output the summaries with the final precision at the end of each verification run.

6 Experimental Evaluation

We implemented our regression verification technique on top of CPAchecker [17]. CPAchecker provides a configurable framework for software verification, with both predicate abstraction and counterexample-guided abstraction refinement supported. To support regression verification, we realized the following functionalities in CPAchecker :

- summary dumping, i.e., dump all summaries to an external file at the end of the verification,
- summary selection, i.e., load and select summaries from an external file at the beginning of the verification (Section 5.2), and
- lazy counterexample analysis (Section 5.3).

Moreover, precision reuse (Section 5.4) was also integrated into our implementation. In the following, we call our enhanced implementation CPAchecker$^+$.

Experiments are designed to answer the following research questions:

- **RQ1.** Is summary reuse efficient enough for regression verification?
- **RQ2.** What is the impact of precision reuse on the efficiency of our approach?
- **RQ3.** What is the impact of summary types on the efficiency of our approach?
- **RQ4.** What is the impact of the lazy counterexample analysis on the efficiency of our approach?

6.1 Experimental Setup

We prepared two industrial benchmarks with 3,675 program revisions of 488 Linux device drivers to evaluate our approach:

1) The first benchmark, obtained from [7], consists of 1,119 real-world program revisions of 62 Linux device drivers.
2) The second benchmark, prepared by ourselves, consists of 2,556 program revisions of 426 Linux device drivers, where all drivers were collected[1] from the "SystemsDeviceDriversLinux64Reach-Safety" category of the 6th International Competition on Software Verification (SV-COMP'17) [28]. For each driver, we make the program in [28] as the base revision, and use a state-of-the-art mutation tool MiLu [29] to randomly generate 5 artificial new revisions.

Recall that all program revisions in the first benchmark were obtained from the official Linux kernel repositories [7], and they are real revisions implemented by the experienced programmers. We use this benchmark to evaluate our approach on real program changes. In contrast, program revisions in the second benchmark were obtained by adding mutations to the base revision of each device driver. An advantage of the second benchmark is that the mutants generated by random pattern can involve much more *unpredictable modifications* than those wirtten by human programmers. We use the second benchmark to evaluate our approach on a broader range of program changes.

All experiments are performed on a machine with Intel Xeon E5-2620 CPU of 2.4GHz 24 cores and 32GB RAM. We use Ubuntu 16.04 (64-bit) with Linux 4.4.0 and jdk1.8.0. The CPAchecker is configured using the *predicateAnalysis-ABE* option. Each verification run is limited to 300 seconds (total CPU time), 6GB of Java heap size and 6 CPU cores.

6.2 Overall Results on Real Revisions (RQ1)

This experiment evaluates our approach on real program revisions. We compare the performance of CPAchecker$^+$ with CPAchecker on the first benchmark. All regression verification techniques, including summary reuse, precision reuse, and lazy counterexample analysis are enabled for CPAchecker$^+$ in this experiment. For simplicity, in the following, we refer to our approach as "Reuse", and the standard CPAchecker as "no Reuse".

In our experiments, a *verification task* is to verify a program revision against a specification. Note that a device driver may have multiple program revisions and also multiple specifications. A pair of a device driver and a specification involves a sequence of verification tasks, where the base revision is verified from scratch, while the other revisions are verification in an incremental way, i.e., as regression verifications. In total, there are 259 driver/specification pairs and 4,193 verification tasks in the first benchmark. Among all tasks, 3,934 are regression verification tasks.

Experimental results are listed in Table 1. Due to page limitation, we restrict this table to the 40 best and 10 worst cases out of the total of 259 driver/specification pairs (sorted by the "Speedup" column). The first two columns ("Driver" and "Specification") list the device driver name and the specification name, respectively. The third column "LoC" shows the lines of code for the base revision of each device driver. The fourth column ("#T") shows the number of regression verification tasks (i.e., the number of revisions minus 1) for each driver/specification pair. The fifth column ("T_{1st}") lists the analysis time for verifying the first revision which is not a regression verification task. This "T_{1st}" gives us the information on the complexity of verifying each device driver.

The following two column assemblies report the experimental results by "no Reuse" and "Reuse" approaches, respectively. For both approaches, we report the number of successfully verified regression tasks "#solved"; the total number of abstract successor computations "#abs_succ"[2] and the total analysis time "T_{rv}" (in seconds) for each driver/specification pair. To conduct a fair comparison, "T_{rv}" and "#abs_succ" are limited to regression verification tasks that are solved by both approaches. The "Speedup" column shows the average speedup of "Reuse" approach

1. The selection strategy is as similar as in [7]. We limit our selection to drivers of Linux 3.4 kernel and with the mutex lock/unlock specifications, and skipped programs whose total CPU time is less than 0.5s and those that need no refinement.

2. An abstract successor is a successor of the current state on the abstract model. Abstract successors computation needs to invoke a SMT solver and is considered as the most time-costly operation in predicate abstraction [23].

over the "no Reuse" approach, calculated by: $1 - T_{rv2}/T_{rv1}$. The last "AvgFSize" and "RSR" columns report the average size (in Kilobytes) of summary files, and the average reusable summary ratio, among all revisions of each driver/specification pair, respectively. For each regression verification task, RSR is the proportion of summaries that are kept after the summary selection.

Recall that each row in the table corresponds to a driver/specification pair. The last two rows ("Sum" and "Avg") take the total and average amount of all rows in the table, respectively.

From Table 1, we observed that our method outperforms "no Reuse" in vast majority of cases. Considering "Speedup" column, among 259 driver/specification pairs, only one pair that our method is slower. Comparing "#solved" columns of both approaches, 216 more regression verification tasks were solved with our approach. This witnesses the value of summary reuse for regression verification.

Among the common 3581 regression verification tasks that both approaches can verify, "no Reuse" takes 151.8 thousand seconds of analysis time while our "Reuse" approach finishes in 10.5 thousand seconds. The overall time speedup of our approach is 93.1%.

Comparing the numbers of abstract successor computations ("#abs_succ.") required by "Reuse" and "no Reuse" for each spec/driver pair, we found that our method cuts down the amount significantly (about 98% reduction), which can explain the reason for the speedup of analysis time.

Let us look at the "AvgFSize" column. The average size of summary files among all regression revisions is 17.4KB (the median is 3.8KB). The added overhead by our approach in storage is acceptable.

Scaling with Larger Changes

The changes between adjacent revisions may not be very significant. We further use the following settings to evaluate our approach on larger changes:

- *4th*: we set up a regression verification task every 4 revisions of the program, and
- *2Revs*: we set up a regression verification task for the last revision of each program.

Finally, we get respectively 898 and 259 regression verification tasks using the above two settings.

Experimental results are listed in Table 2. The original setting that incrementally verifies each adjacent revision of the program is referred to as *All* in the table. The average numbers of changed lines for regression verification tasks using the above three settings are 511, 1242 and 1562, respectively. The increasing *avg. diff. lines* lead to decreasing *RSR*, which is reasonable since more program changes can of course lead less summaries to be reusable.

Observe that our approach can still get considerable performance improvements (86.1% and 80.8% of speedups) with the *4th* and *2Revs* settings, which show the effectiveness of our approach on larger changes.

6.3 Overall Results on Artificial Revisions (RQ1)

The second experiment evaluates our approach on artificial program revisions in the second benchmark. This experiment contains 2,556 verification tasks, involving 426 driver/specification pairs. Among all tasks, 2,130 are regression verification tasks.

Results of this experiment are listed in Table 3. Again, we limit this table to the 40 best and 10 worst cases out of all 426 driver/specification pairs (sorted by the "Speedup" column). Each column is with the same meaning as in Table 1. Note that there is only one specification for each driver in this benchmark, the "Spec." column is thus skipped.

From this table, we observed similar results as in Table 1. Among all 426 driver/specification pairs, our approach wins on 389 pairs. In total, our approach solved 10 more verification tasks, and the average speedup is 84.2%.

6.4 Comparison with Existing Tools (RQ1)

To further demonstrate the efficiency of our approach, two more experiments were conducted to compare CPAchecker$^+$ with the existing regression verification tools:

- eVolcheck [30], a regression verification tool that implements the technique of interpolation-based procedure summaries [11], and
- UAutomizer$^+$ [31], a regression verification extension of the famous software verification tool UAutomizer [19].

6.4.1 Comparison with eVolcheck

This experiment was conducted on the set of real-world programs. Before this experiment, some of the programs need to be modified to adapt to the input format of eVolcheck, e.g., replacing "ldv_error()" by "assert(0)".

Note that eVolcheck [30] is just an experimental implementation, and is not fully optimized. Among all 4,193 verification tasks, eVolcheck failed on 3,646 tasks due to various parsing and runtime errors. The comparative experiment was conducted on the remaining 547 verification tasks.

The comparison results are listed in Table 4, where the T_{rv1} and T_{rv2} columns report the total regression verification time *without* and *with* reuse, respectively, and the "Speedup" is calculated by $1 - T_{rv2}/T_{rv1}$. Note that eVolcheck employs the bounded model checking technique, and its unwinding factor was set to 15. Among the 547 verification tasks, eVolcheck solved (i.e., the underlying SMT solver returned a result) 143 tasks within the time limit of 300 seconds, whereas our CPAchecker$^+$ solved all. In comparison of the efficiency of the employed reuse techniques, the speedups of eVolcheck and CPAchecker$^+$ are 75.2% and 89.7%, respectively. These results demonstrate the efficiency of our summary reuse technique.

6.4.2 Comparison with UAutomizer

This experiment was conducted on the real-world benchmark, too. Excluding the programs that UAutomizer fails to parse, there are totally 1,177 verification tasks that belong to 90 driver/specification pairs. [3].

Note that the adopted verification techniques are very different in these two tools: UAutomizer$^+$ uses the trace

3. In their original paper [31], the UAutomizer$^+$ was evaluated on the same set of programs.

TABLE 1
Overall experimental results on real revisions

Run set					no Reuse			Reuse			Speed	Avg	
Device	Spec.	LoC	#T	T_{1st}	#solved	#abs_succ	T_{rv1}	#solved	#abs_succ	T_{rv2}	-up	FSize	RSR
wm831x	39_7a	5183	34	61.9	34	346.7K	144.0	34	0.2K	1.4	99.0%	7.2	0.8
cx231xx	08_1a	8241	13	21.0	11	356.4K	203.9	13	0.1K	1.8	99.0%	1.5	0.8
wm831x	32_7a	4482	34	2.2	34	253.8K	125.1	34	0.2K	1.5	98.8%	5.7	0.8
usb-az6007	08_1a	7912	5	60.6	5	250.4K	94.5	5	0.2K	1.2	98.7%	2.5	0.7
wm831x	08_1a	4579	34	50.2	34	178.1K	103.2	34	0.2K	1.3	98.7%	4.1	0.8
abyss	68_1	5382	2	82.6	2	41.8K	35.6	2	0.0K	0.5	98.7%	2.6	0.8
abyss	32_1	4251	3	65.1	3	46.6K	46.9	3	0.0K	0.6	98.7%	2.3	0.8
wm831x	68_1	6569	3	64.7	3	164.9K	73.0	3	0.1K	1.0	98.6%	4.4	0.9
leds-bd2802	32_1	6346	4	57.0	4	232.1K	84.7	4	0.1K	1.2	98.5%	8.5	0.9
cp210x	68_1	8169	14	44.2	10	234.1K	102.8	14	0.3K	1.1	98.5%	3.3	0.9
wm831x	32_1	5339	3	49.7	3	121.9K	61.2	3	0.1K	0.9	98.5%	4.1	0.9
spcp8x5	68_1	6086	13	68.0	12	156.2K	86.1	13	0.2K	1.3	98.4%	4.6	0.9
mos7840	08_1a	8971	60	81.0	28	270.6K	156.1	60	0.3K	1.3	98.2%	6.1	0.6
spcp8x5	32_1	6002	13	52.5	13	112.8K	72.3	13	0.2K	1.3	98.2%	4.3	0.9
sil164	39_7a	7026	3	54.2	3	74.3K	33.8	3	0.1K	0.6	98.2%	8.7	0.8
cx231xx	39_7a	9959	13	61.5	2	67.3K	29.7	13	0.0K	0.1	98.1%	12.0	0.6
xilinx_uartps	32_7a	5393	3	75.2	3	47.0K	45.6	3	0.1K	1.0	97.9%	3.7	0.8
tcm_loop	39_7a	12515	41	52.4	41	186.9K	72.7	41	0.2K	1.6	97.8%	16.1	0.9
xilinx_uartps	08_1a	5239	3	72.0	3	46.8K	44.3	3	0.1K	1.0	97.7%	3.3	0.8
catc	32_1	6245	10	25.2	10	39.1K	40.0	10	0.1K	0.9	97.6%	3.0	0.9
sil164	32_7a	6805	3	53.6	3	68.3K	43.8	3	0.2K	1.1	97.5%	7.3	0.8
ems_usb	39_7a	9647	21	70.3	21	175.9K	70.2	21	0.2K	1.7	97.5%	11.3	0.7
tdo24m	39_7a	5529	12	71.4	12	220.7K	107.1	12	0.7K	2.7	97.5%	6.5	0.6
i915	68_1	13916	79	37.9	79	26.7K	41.2	79	0.0K	1.1	97.3%	1.8	1.0
cp210x	39_7a	6909	71	79.8	56	300.1K	116.2	71	0.9K	2.5	97.3%	8.2	0.6
ssu100	08_1a	5985	28	41.9	28	54.4K	45.5	28	0.2K	1.4	97.0%	3.8	0.8
i2o_scsi	08_1a	5644	7	42.3	7	27.1K	35.8	7	0.1K	1.1	97.0%	2.1	0.7
i2o_scsi	39_7a	7515	6	65.5	6	80.0K	49.6	6	0.2K	1.5	97.0%	8.5	0.7
i915	32_1	13557	79	30.8	79	18.5K	35.7	79	0.0K	1.1	97.0%	1.6	1.0
cp210x	08_1a	8370	71	183.6	64	167.6K	94.4	71	0.9K	2.6	96.9%	2.9	0.6
budget-patch	39_7a	8446	9	105.8	1	265.9K	105.8	8	0.1K	0.4	96.9%	11.3	0.8
tty-serial	39_7a	6216	9	81.8	9	135.1K	66.8	9	0.2K	2.1	96.9%	17.4	0.8
spcp8x5	08_1a	5751	37	49.8	37	118.0K	62.5	37	0.3K	2.1	96.7%	3.7	0.7
ems_usb	32_1	8646	6	30.7	6	24.0K	26.0	6	0.1K	0.9	96.6%	4.7	1.0
i915	39_7a	14298	79	49.5	79	70.7K	60.9	79	0.3K	2.1	96.6%	8.3	0.9
catc	08_1a	5878	22	23.4	22	51.8K	42.5	22	0.2K	1.5	96.5%	3.6	0.8
cx231xx	32_7a	8393	13	22.5	2	5.8K	12.1	4	0.0K	0.2	96.4%	4.5	0.6
wl12xx_sdio	39_7a	8578	38	67.4	38	120.6K	60.6	38	0.5K	2.2	96.4%	32.9	0.9
abyss	08_1a	4136	4	26.2	4	17.6K	20.5	4	0.0K	0.7	96.4%	1.8	0.6
catc	39_7a	7941	22	46.6	22	537.7K	182.1	22	3.7K	6.8	96.3%	29.8	0.8
⋮						⋮					⋮		
i2c-algo-pca	32_7a	2879	14	0.8	14	0.4K	1.9	14	0.1K	0.9	50.0%	1.7	0.8
mtd-mtdoops	43_1a	2898	20	1.6	20	0.2K	1.9	20	0.1K	1.0	50.0%	2.9	0.7
mtd-ar7part	32_1	1003	2	1.2	2	0.1K	0.7	2	0.0K	0.3	49.6%	0.7	0.7
wm831x	43_1a	4246	3	46.1	3	64.6K	43.5	3	20.6K	26.0	40.3%	18.8	0.9
farsync	32_7a	6823	9	2.4	9	0.8K	2.9	9	0.4K	1.8	38.0%	5.8	0.7
drbd	08_1a	57282	96	3.4	96	0.1K	5.2	96	0.0K	3.3	35.6%	0.6	1.0
i2c-algo-pca	43_1a	3031	7	0.9	7	0.1K	1.5	7	0.1K	1.2	18.4%	1.0	0.4
vmalloc	32_7a	3885	29	11.0	29	3.4K	9.9	29	0.8K	8.2	17.3%	1548.7	0.6
paride-pt	32_7a	5163	9	21.3	9	13.4K	18.9	9	13.3K	17.4	8.0%	3.2	0.4
mtd-ar7part	43_1a	1000	2	0.5	2	0.0K	0.6	2	0.0K	0.6	-8.9%	0.9	0.5
Sum	-	1.5M	3934	6.0K	3588	228.3M	151.8K	3804	3.2M	10.5K	-	-	-
Avg	-	6128	15.2	23.3	13.9	881.4K	586.0	14.7	12.5K	40.5	93.1%	17.4	0.8

abstraction, while our CPAchecker$^+$ uses the predicate abstraction. Moreover, the regression verification techniques implemented in these two tools are also different: one attempts to reuse the previously generated Floyd-Hoare automata [31], while another attempts to reuse the previously generated state-based summaries. To compare the efficiency of their adopted regression verification techniques, we compare the speedups of these two tools (with reuse over without reuse).

The comparison results are listed in Table 5. Note that UAutomizer$^+$ implements two reuse strategies, i.e., *Eager* and *Lazy*. Results for both strategies are reported. From Table 5, CPAchecker$^+$ achieves a speedup of 90.8%, and UAutomizer$^+$ gets a speedup of 82.8% or 81.4%. This result demonstrates the efficiency of our summary reuse technique. Note that we cannot conclude the superiority of our reuse technique over UAutomizer$^+$ from this result, since they are used in different verification frameworks.

6.5 Impact of Reuse Strategies (RQ2)

In the former two experiments, both summary reuse and precision reuse were enabled for our approach. In this experiment, we switch off "precision reuse" and "summary

TABLE 2
Overall experimental results on larger changes

Revs.	#T	Avg. Diff. Lines	no Reuse #solved	no Reuse T_{rv1}	Reuse #solved	Reuse T_{rv2}	Speed -up	RSR
All	3934	511	3588	151.8K	3804	10.5K	93.1%	0.8
4th	898	1242	812	32.7K	870	4.5K	86.1%	0.6
2Revs	259	1562	240	7.7K	245	1.5K	80.8%	0.4

reuse" respectively, and evaluate the efficiency of our approach under different reuse strategies, i.e., "no Reuse", "Precision Reuse", "Summary Reuse" and "both Reuse". This experiment was conducted on the second benchmark.

Table 6 shows the results of this experiment. Every row sums up results of all 2,130 regression verification tasks. For each row, we report the total regression verification time "T_{rv}" (in seconds), the total number of solved regression tasks "#solved", and the average memory usage "Mem" (in gigabytes). Note that we only list the time usage in "no Reuse" row. For the other three rows, we show the speedup against "no Reuse" approach.

From Table 6, we found that every reuse technique outperforms "no Reuse". They are not surprising since the precision reuse can significantly reduce the CEGAR iterations [7], and thus cuts down the verification time (73.5% speedup); and the summary reuse can save the repeated computation of summaries, and thus also reduces the verification time (69.5% speedup). Moreover, "Summary Reuse" solves 2 more tasks than "Precision Reuse", illustrating that the former technique is more robust than the latter one. Precision reuse and summary reuse are two orthogonal techniques. By integrating these two techniques, "both Reuse" get the best performance, not only in analysis time (84.2% speedup), but also in the number of verified regression tasks.

The "Mem" column shows that all reuse strategies save the memory usage meetly. Again, "both Reuse" saves the most on memory consumption.

6.6 Impact of Summary Types (RQ3)

This experiment investigates the efficiency of our approach on different types of summaries. We evaluate the efficiency of our approach with loop summaries reused, procedure summaries reused and all summaries reused, respectively. Note that "precision reuse" and "lazy counterexample analysis " are switched off in this experiment.

We accumulate the analysis time on different summary types. Results are illustrated in Fig 7, where the X-axis indicates the number of device drivers, and the Y-axis represents the accumulated analysis time. From Fig 7, we observed that "Procedure Summaries" outperforms "Loop Summaries", and "All Summaries Reuse" performs the best. The main reason is that our benchmark contains fewer loop statements than procedures (1,273 versus 11,417).

6.7 Impact of Lazy Counterexample Analysis (RQ4)

The final experiment evaluates the efficiency of lazy counterexample analysis. Note that this technique is mainly relative to the refinement process, we measure the refinement time and counterexample length in this experiment.

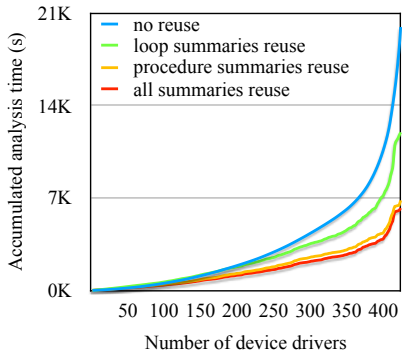


Fig. 7. Accumulating the analysis time on different summary types

Fig. 8(a) shows the results on counterexample length in a scatter diagram. Note that each device driver may involve several regression verification tasks, and each regression verification task may require many CEGAR iterations. The reported counterexample length is the accumulated length of all counterexamples generated in all CEGAR iterations among all regression verification tasks of a device driver. In Fig. 8(a), each point represents the accumulated counterexample length of a device driver, the Y and X axes indicate our approach with and without lazy counterexample analysis, respectively. Both X and Y axes are logarithmic. A point below the reference line indicates a case where the lazy counterexample analysis is beneficial.

Fig. 8(b) show results on refinement time. Again, the reported refinement time is the accumulation of time spent on all refinement iterations among all regression verification tasks of all the currently-tested device drivers. In Fig. 8(b), the X-axis catalogs the number of device drivers and the Y-axis shows the accumulated refinement time.

On the whole, the lazy counterexample analysis performs better on refinement, saving about 29.9% of time. It is also observed that for 415 of 426 device drivers, the corresponding data points in Fig. 8(a) are above the reference line, indicating that the accumulated counterexample lengths were reduced with this technique. These results conform to our algorithmic analysis in Section 5.3. With the lazy analysis technique, the counterexample is not necessarily to be fully expanded. And a shorter counterexample can usually reduce the refinement efforts.

7 Related Work

Regression verification was investigated mainly in two directions, the verification of differences, and the reuse of previously computed results. We also discuss the summarization and symbolic execution techniques in this section.

TABLE 3
Overall experimental results on artificial revisions

Run set				no Reuse			Reuse			Speed	Avg	
Device	LoC	#T	T_{1st}	#solved	#abs_succ	T_{rv1}	#solved	#abs_succ	T_{rv2}	-up	FSize	RSR
iuu_phoenix	10258	5	214.5	5	136306	202.2	5	105	2.1	99.0%	4.3	0.9
broadcom1	4931	5	75.6	5	244123	72.0	5	77	1.1	98.5%	2.3	0.9
sungem_phi	5287	5	184.3	5	249324	115.2	5	270	2.0	98.3%	3.6	0.9
paride-bpck	9896	5	89.4	5	85737	89.1	5	106	1.7	98.1%	3.8	0.9
rtl2830	4457	5	111.9	5	16172	107.9	5	69	2.3	97.9%	2.4	0.9
mwifiex_pcie	11367	5	95.1	5	128217	99.0	5	164	2.3	97.6%	7.2	0.9
paride-epat	7695	5	106.1	5	22426	103.4	5	271	2.7	97.4%	2.9	0.8
tty-serial-mfd	10237	5	67.1	5	135512	66.4	5	120	1.9	97.2%	9.3	1.0
paride-kbic	8009	5	77.9	5	88482	74.9	5	190	2.2	97.1%	3.0	0.9
tpm_nsc	4791	5	35.5	5	10807	36.2	5	69	1.1	97.1%	2.7	0.9
mxl111sf-demod	8288	5	32.2	5	24957	31.7	5	41	1.0	96.7%	3.0	0.9
paride-on26	17745	5	134.5	5	80637	137.1	5	443	5.0	96.4%	2.7	0.9
dib3000mb	7868	5	120.3	5	26216	123.1	5	447	4.6	96.2%	4.0	0.9
paride-on20	7747	5	40.9	5	20099	41.3	5	106	1.6	96.1%	1.5	0.9
paride-dstr	6604	5	34.3	5	16655	33.8	5	84	1.4	95.8%	2.4	0.9
cmd640	6406	5	38.5	5	36496	37.7	5	93	1.6	95.6%	4.7	0.9
it913x	5947	5	69.1	5	14337	53.6	5	82	2.4	95.6%	3.0	0.9
serqt_usb21	10282	5	56.8	5	53423	56.4	5	166	2.6	95.3%	7.3	1.0
tuners-qt1010	7043	5	54.1	5	2645	47.9	5	79	2.3	95.3%	2.0	0.8
cfag12864b	1493	5	15.0	4	5278	14.8	5	13	0.6	95.1%	8.8	1.0
paride-comm	5408	5	24.9	5	13211	23.8	5	77	1.2	94.9%	2.5	0.9
cp210x	7113	5	186.9	5	274622	166.8	4	1526	10.8	94.8%	14.6	0.9
paride-ktti	2886	5	15.1	5	6299	15.3	5	52	0.8	94.7%	1.5	0.8
paride-friq	8979	5	40.8	5	12680	40.1	5	146	2.2	94.6%	2.9	0.9
google-gsmi	6826	5	20.8	5	5294	19.9	5	48	1.1	94.3%	3.8	0.9
spcp8x5	6913	5	29.6	5	17396	23.3	5	50	1.3	94.3%	3.5	1.0
i2c-diolan	5308	5	30.2	5	13085	30.8	5	200	1.8	94.1%	3.4	0.8
sil164	6298	5	16.8	5	6906	16.6	5	66	1.0	94.0%	3.3	0.9
metro-usb	6288	5	25.3	5	20810	25.0	5	73	1.5	94.0%	2.4	0.9
paride-fit3	4459	5	19.5	5	10691	19.7	5	72	1.2	93.9%	1.7	0.9
wm831x-ldo	3720	5	19.9	5	15462	18.7	5	129	1.2	93.3%	4.0	0.9
paride-fit2	2593	5	13.6	5	5301	13.1	5	50	0.9	93.3%	1.5	0.9
sdricoh_cs	1633	5	12.4	5	7725	13.2	5	38	0.9	93.2%	3.5	1.0
tdo24m	5039	5	30.9	5	12666	26.7	5	121	1.9	93.0%	2.5	0.8
hid-zydacron	3211	5	10.6	5	2855	10.2	5	40	0.7	92.9%	2.0	0.9
max8903	1736	5	10.2	5	2984	10.3	5	27	0.7	92.8%	1.4	0.9
paride-epia	5518	5	28.1	5	10999	27.5	5	187	2.0	92.8%	2.8	0.9
wm831x-dcdc	3420	5	21.7	5	18191	22.7	5	149	1.6	92.8%	4.2	1.0
paride-frpw	8046	5	33.2	5	9374	33.4	5	182	2.6	92.2%	3.0	0.9
leds-lp5521	5443	5	6.4	5	526	6.4	5	12	0.6	91.4%	1.9	1.0
⋮				⋮				⋮				
vp3054-i2c1	5582	5	0.6	5	37	0.6	5	17	0.9	-40.1%	0.5	0.9
atlas_btns1	3599	5	0.6	5	30	0.6	5	20	0.8	-40.7%	0.5	0.9
girbil-sir1	5260	5	0.8	5	29	0.8	5	25	1.1	-42.1%	0.5	0.9
sbc8360	2052	5	0.6	5	97	0.6	5	29	0.9	-45.5%	1.1	0.9
vivopay	3793	5	0.5	5	11	0.5	5	5	0.8	-49.0%	0.4	0.8
epx_c31	2435	5	0.6	5	115	0.7	5	50	1.0	-49.3%	1.0	0.9
vgg2432a41	4136	5	0.7	5	41	0.6	5	33	0.9	-50.7%	0.5	0.9
siemens_mpi	3788	5	0.5	5	11	0.5	5	5	0.7	-54.6%	0.4	0.8
icplus1	4968	5	0.5	5	20	0.5	5	12	0.8	-76.3%	0.3	1.0
vp7045	8254	5	10.8	5	1407	11.8	5	270	26.2	-121.9%	61.3	0.9
Sum	1828793	2130	4375.5	2119	13.8M	20.9K	2129	247847	3.3K	-	-	-
Avg	4293	5	10.3	5	32.5K	49.1	5	581.8	7.7	84.2%	3.1	0.9

TABLE 4
Comparison of our approach with eVolcheck

	#computable	T_{rv1}	T_{rv2}	Speedup
eVolcheck	143	2382.7	591.9	75.2%
CPAchecker$^+$	244	2074.2	213.2	89.7%

TABLE 5
Comparison of our approach with UAutomizer$^+$

	Speedup
CPAchecker$^+$	90.8%
UAutomizer$^+$-Eager	82.8%
UAutomizer$^+$-Lazy	81.4%

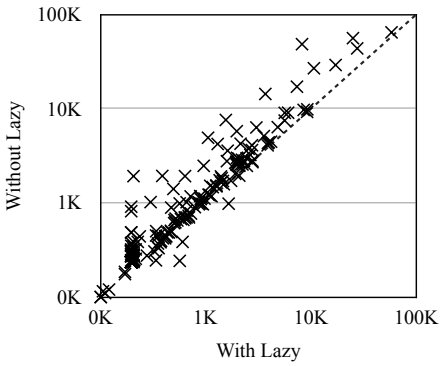
Verification of Differences

In this line of research, one attempts to establish the correctness of the new program by proving its (conditional) equivalence to an old and verified program.

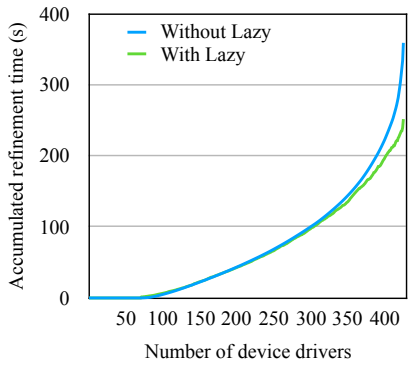
Many techniques have been proposed in this line of research. The technique for proving conditional equivalence of two programs by abstraction and decomposition of pro-

TABLE 6
Results on different reuse strategies

	T_{rv}	#solved	Mem
no Reuse	20898.8	2119	589
Precision Reuse	73.5%	2125	151
Summary Reuse	69.5%	2127	180
Both Reuse	84.2%	2129	137



(a) Accumulated counterexample length



(b) Accumulated refinement Time

Fig. 8. Performance of summary reuse with and without *lazy counterexample analysis*

cedures is proposed in [4], [32]. Backes et al. [5] proposed to distinguish the program behaviors that are impacted by the changes. Only the impacted program behaviors needed to be considered during the regression verification. Beyer et al. [33] proposed the conditional model checking, which outputs a condition such that the program satisfies the specification under this condition. Böhme et al. [6] proposed a partition-based regression verification technique. Instead of proving the absence of regression errors for the entire input space, this approach continuously verifies the input space in a gradual manner. Felsing et al. [34] reduced the equivalence proving of two related imperative integer programs to Horn constraints over uninterpreted predicates, and then solved the constraints using an Horn solver.

Moreover, Rungta et al. [35] presented a technique for interprocedural change impact analysis. Yang et al. [36] introduced an incremental approach for checking the conformance of code against different properties. Trostanetski et al. [37] analyzed the semantic difference between successive revisions. Mora et al. [38] performed modular symbolic execution to prove the equivalence between different versions of libraries with respect to the same parts of codebase (client program).

Reuse of Intermediate Results

In this line of research, one studies the reuse of previously-generated results to the current verification. A variety of information has been proposed for reuse.

Some researchers [9], [39], [40], [41] proposed to keep the reached state space and reuse them in the further verification runs. The rationale of these techniques is that state spaces of consecutive versions tend to be similar. However, recording and reusing reached state space may be costly, and these techniques may not be applicable to large-scale programs. For example, [40] points out 6 times more on memory usage in the worst case. The lines of code of single revision of [40] are less than 1,000.

Visser et al. [10] noticed the importance of constraint solving for symbolic execution. They proposed to cache and reuse the results of constraint solving. This approach was further improved in [42], [43] from different aspects. This group of techniques is orthogonal to our approach. These techniques can be applied to enhance our approach.

Beyer et al. [7] proposed to use abstract precisions as the intermediate results. An abstract precision defines the level of abstraction, which conveys important information on the current verification. They proposed to record the final abstract precision and to reuse it as the initial abstract precision of the current verification. With this technique, the number of refinements can often be reduced. Note that the precision reuse and our summary reuse are orthogonal to each other. It is possible to combine these two reuse techniques together. We have already combined this technique with ours. The combined technique shows a very promising performance.

Fedyukovich et al. [44] offered a regression verification technique for checking property directed equivalence. The safe inductive invariants across program transformations were migrated and established. Rothenberg et al. [31] proposed to reuse the sequence of Floyd-Hoare automata learned during the trace abstraction. Two reuse strategies, eagerly and lazily, were developed in this paper. This technique has been realized in UAutomizer, a well-known software verification tool.

The work most relevant to ours is [11], [30], where a regression verification technique by means of interpolation-based procedure summaries was proposed. Our idea of summary reuse was inspired by these two papers. The main difference lies in the way in which the summaries are constructed during the verification. In [11], [30], the authors use a logical formula φ_A to encode the behaviors of the procedure ϱ, and another logical formula φ_B to encode its calling context. Then they compute the interpolation of φ_A and φ_B and use that as the summary of ϱ. In contrast, we use the abstract states in predicate analysis to construct the program summaries. Each summary in our paper consists of an entry state and a set of exit states of ϱ. Our state-based summaries can be completely integrated into the framework of CEGAR. All ingredients of a state-based summary are by-products of CEGAR. There needs no additional computation for generating this kind of summaries. Experimental

comparison in Section 6.4 demonstrates the practicability of our approach.

Pastore et al. [45] proposed a method to validate that an already tested code has not been broken by an upgrade. It maintains a test suite that can be used to revalidate the software as it evolves. Different from our approach, this technique is respect to regression testing. The verification technique is used there, as an aid, to validate dynamic properties (or invariants). In contrast, we aim to provide a new regression verification technique via reusing summaries.

Summarization

In this line of research, one tries to replace program fragments with summaries. A summary can usually be represented as an input-output pair of a program fragment. Procedure summaries have been long studied and there are also many studies on loop summaries recently.

Many researchers [11], [30], [46], [47] proposed to use interpolation-based method to generate procedure summaries. [47] combines function summaries with the expressiveness of satisfiability modulo theories (SMT), which makes summaries smaller and more human-readable. [11], [30], [46] implement a procedure summarization approach for software bounded model checking, and uses interpolation-based procedure summaries as over-approximation of procedure calls.

Kroening et al. [48] proposed the idea to substitute a loop with a conservative abstraction of its behavior, constructing abstract transformers for nested loops starting from the inner-most loop. They also applied this method in termination analysis. Seghir el al. [49] used various inference rules for deriving summaries based on control structures. However, this approach can only compute precise loop summaries for restricted classes of programs depending on inference rules. Xie et al. [50] proposed a general framework for summarizing multi-path loops. It classifies loops according to the patterns of values changes in path conditions and the interleaving of paths within the loop. A disjunctive summarization is constructed for all the feasible executions in the loop. Different from our method, [50] cannot summarize loops containing non-induction variables, array variables, and nested loops. Godefroid et al. [51] investigated an alternative approach based on automatic loop-invariant generation. This approach can (partially) summarize a loop body during a single dynamic symbolic execution, which can ease the path explosion in dynamic test generation.

Symbolic Execution

In recent years, a great deal of effort has been focused on regression symbolic execution, which takes advantage of the previous analysis of symbolic execution to speedup the current analysis.

Person et al. [52] used a form of overapproximating symbolic execution to skip portions of the program that are provably identical across the versions. In [53], Person et al. presented a regression symbolic execution technique for Java programs, based on the Symbolic PathFinder. It analyzes the CFAs of two program versions, computes the locations affected by the program changes, and then applies the symbolic execution to the affected code only. Further more, Guo et al. [54] investigated the symbolic execution technique for multi-threaded programs.

8 Conclusion

We proposed in this paper a fully automatic regression verification technique in the context of CEGAR. Abstract summaries are reused across different abstract precisions and different program revisions. We proposed a unified framework for reusing both procedure summaries and loop summaries. A lazy counterexample analysis algorithm was further proposed to reduce the unnecessary path expansion efforts. We implemented our approach in the software verification tool CPAchecker. Experimental results show the promising performance of our technique.

9 Acknowledgment

This work was supported in part by the National Natural Science Foundation of China (61672310, 61527812), the National Key R&D Program of China (2018YFB1308601).

References

[1] V. D'silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[2] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 2, pp. 184–208, 2001.

[3] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529–551, 1996.

[4] B. Godlin and O. Strichman, "Regression verification," in *Proceedings of the 46th Annual Design Automation Conference*. ACM, 2009, pp. 466–471.

[5] J. Backes, S. Person, N. Rungta, and O. Tkachuk, "Regression verification using impact summaries," in *Model Checking Software*. Springer, 2013, pp. 99–116.

[6] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "Partition-based regression verification," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 302–311.

[7] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, "Precision reuse for efficient regression verification," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 389–399.

[8] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, "Differential assertion checking," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 345–355.

[9] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido, "Extreme model checking," in *Verification: Theory and Practice*. Springer, 2003, pp. 332–358.

[10] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 58.

[11] O. Sery, G. Fedyukovich, and N. Sharygina, "Incremental upgrade checking by means of interpolation-based function summaries," in *Formal Methods in Computer-Aided Design (FMCAD), 2012*. IEEE, 2012, pp. 114–121.

[12] M. Sharir and A. Pnueli, *Two approaches to interprocedural data flow analysis*. New York University. Courant Institute of Mathematical Sciences. ComputerScience Department, 1978.

[13] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, pp. 49–61.

[14] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 3, p. 16, 2007.

[15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer, 2000, pp. 154–169.

[16] D. Beyer, "Reliable and reproducible competition results with BenchExec and witnesses (report on sv-comp 2016)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 887–904.

[17] D. Beyer and M. E. Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification*. Springer, 2011, pp. 184–190.

[18] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[19] M. Heizmann, J. Hoenicke, and A. Podelski, "Software model checking for people who love automata," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 36–52.

[20] M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi, "SMACK software verification toolchain," in *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering (ICSE) Companion*, W. Visser and L. Williams, Eds. ACM, 2016, pp. 589–592.

[21] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT press, 1999.

[22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 58–70, 2002.

[23] D. Beyer, T. A. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *CAV*, vol. 7. Springer, 2007, pp. 504–518.

[24] S. Graf and H. Saïdi, "Construction of abstract state graphs with pvs," in *International Conference on Computer Aided Verification*. Springer, 1997, pp. 72–83.

[25] K. L. McMillan, "Interpolation and sat-based model checking," in *International Conference on Computer Aided Verification*. Springer, 2003, pp. 1–13.

[26] ——, "Lazy abstraction with interpolants," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 123–136.

[27] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[28] D. Beyer, "Software verification with validation of results (report on sv-comp 2017)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 331–349.

[29] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference*. IEEE, 2008, pp. 94–98.

[30] G. Fedyukovich, O. Sery, and N. Sharygina, "evolcheck: Incremental upgrade checker for c," in *19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer. Springer, 2013.

[31] B.-C. Rothenberg, D. Dietsch, and M. Heizmann, "Incremental verification using trace abstraction," in *International Static Analysis Symposium*. Springer, 2018, pp. 364–382.

[32] S. Chaki, A. Gurfinkel, and O. Strichman, "Regression verification for multi-threaded programs," in *Verification, model checking, and abstract interpretation*. Springer, 2012, pp. 119–135.

[33] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: a technique to pass information between verifiers," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 57.

[34] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 349–360.

[35] N. Rungta, S. Person, and J. Branchaud, "A change impact analysis to characterize evolving program behaviors," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 109–118.

[36] G. Yang, S. Khurshid, S. Person, and N. Rungta, "Property differencing for incremental checking," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1059–1070.

[37] A. Trostanetski, O. Grumberg, and D. Kroening, "Modular demand-driven analysis of semantic difference for program versions," in *International Static Analysis Symposium*. Springer, 2017, pp. 405–427.

[38] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 441–451.

[39] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 291–300.

[40] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 115–124.

[41] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *International Conference on Computer Aided Verification*. Springer, 2005, pp. 449–461.

[42] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 305–315.

[43] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," *arXiv preprint arXiv:1501.07174*, 2015.

[44] G. Fedyukovich, A. Gurfinkel, and N. Sharygina, "Property directed equivalence via abstract simulation," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 433–453.

[45] F. Pastore, L. Mariani, A. E. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad, "Verification-aided regression testing," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 37–48.

[46] O. Sery, G. Fedyukovich, and N. Sharygina, "Funfrog: bounded model checking with interpolation-based function summarization," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2012, pp. 203–207.

[47] L. Alt, S. Asadi, H. Chockler, K. E. Mendoza, G. Fedyukovich, A. E. Hyvärinen, and N. Sharygina, "Hifrog: Smt-based function summarization for software verification," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2017, pp. 207–213.

[48] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger, "Loop summarization using state and transition invariants," *Formal Methods in System Design*, vol. 42, no. 3, pp. 221–261, 2013.

[49] M. N. Seghir, "A lightweight approach for loop summarization," in *International Symposium on Automated Technology for Verification and Analysis*. Springer, 2011, pp. 351–365.

[50] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 61–72.

[51] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 23–33.

[52] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 226–237.

[53] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Acm Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 504–515.

[54] S. Guo, M. Kusano, and C. Wang, "Conc-ise: Incremental symbolic execution of concurrent software," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 531–542.

Fei He received the B.S. degree in computer science and technology from National University of Defense Technology in 2002, and the Ph.D. degree in computer science and technology from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include formal verification and program analysis.

Qianshan Yu received the B.S. degree from Jilin University in 2017. He is currently a PhD student in the School of Software at Tsinghua University. His research interests include software verification and regression verification.

Liming Cai received the B.S. degree from Xiamen University in 2013, and the M.S. from Tsinghua University in 2016. He is currently an algorithm engineer in kwai tech.co. His research interests include formal methods and program verification.